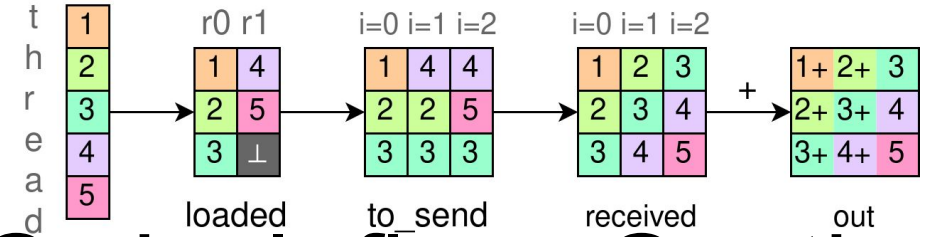
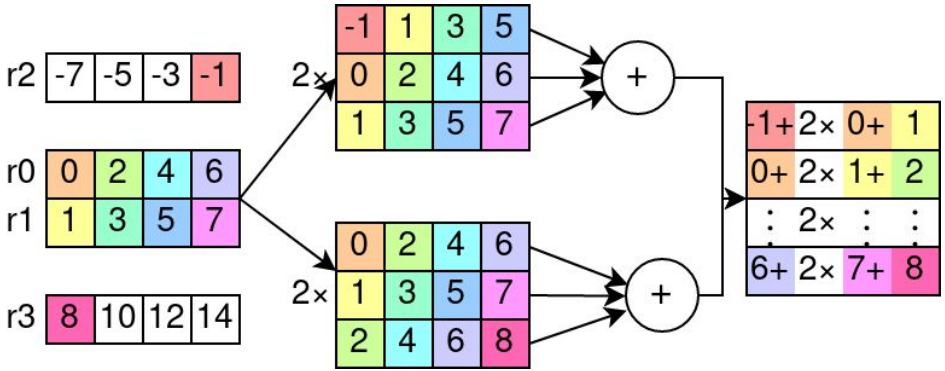


Swizzleflow: Synthesis of Irregular Data Mappings in Accelerator Kernels Using Novel Pruning Abstractions

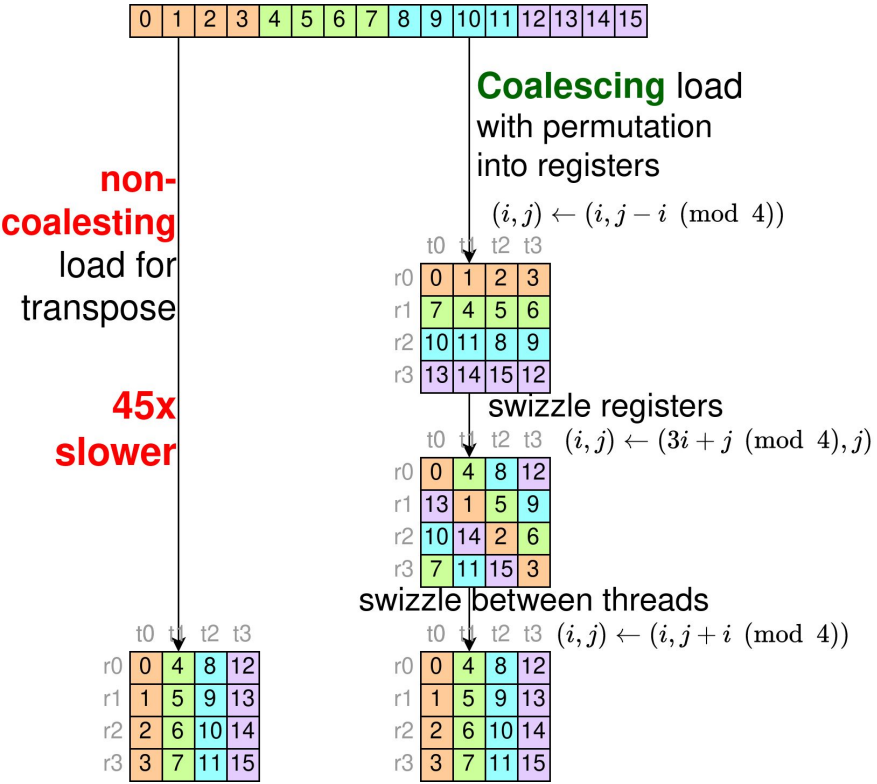
Krzysztof Drewniak with Rastislav Bodik
University of Washington Qualifying Examination

Accelerators constrain programs

- Accelerators = GPUs, vector processors, FPGAs; used for fast math
- Obeying hardware limits → often dramatic speedups
- Data movement limits: ex. parallel banks or coalescing loads on GPUs
- Special instructions require data or problem in a particular form
  - Ex. HVX processor Gaussian filter — 2x faster than Halide



Accelerator constraints: GPU Transpose



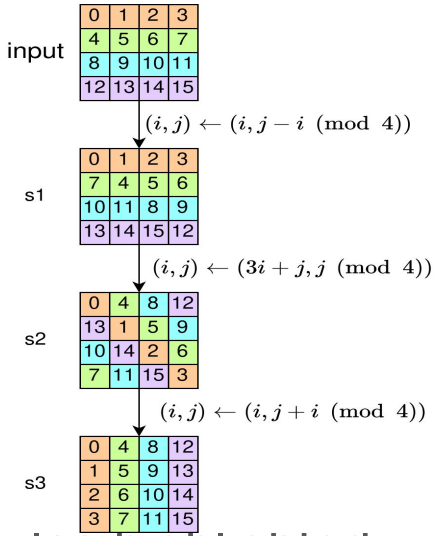
- Coalescing loads stall the GPU less
- Fast algorithm uses *swizzles* — irregular mappings of data to memories or compute elements
- Give dramatic speedups even though algorithm is more complex
- We synthesized this, improving on previous work (Trove, PPoP ‘14)

Modeling accelerator kernels

New language to represent executions of swizzle kernels portably (as shuffling arrays around) — portable representation of accelerator code

```
for (int j = 0; j < T; j++)
  for (int i = 0; i < 4; i++)
    out[j][i] = input[i][j]
```

Transpose

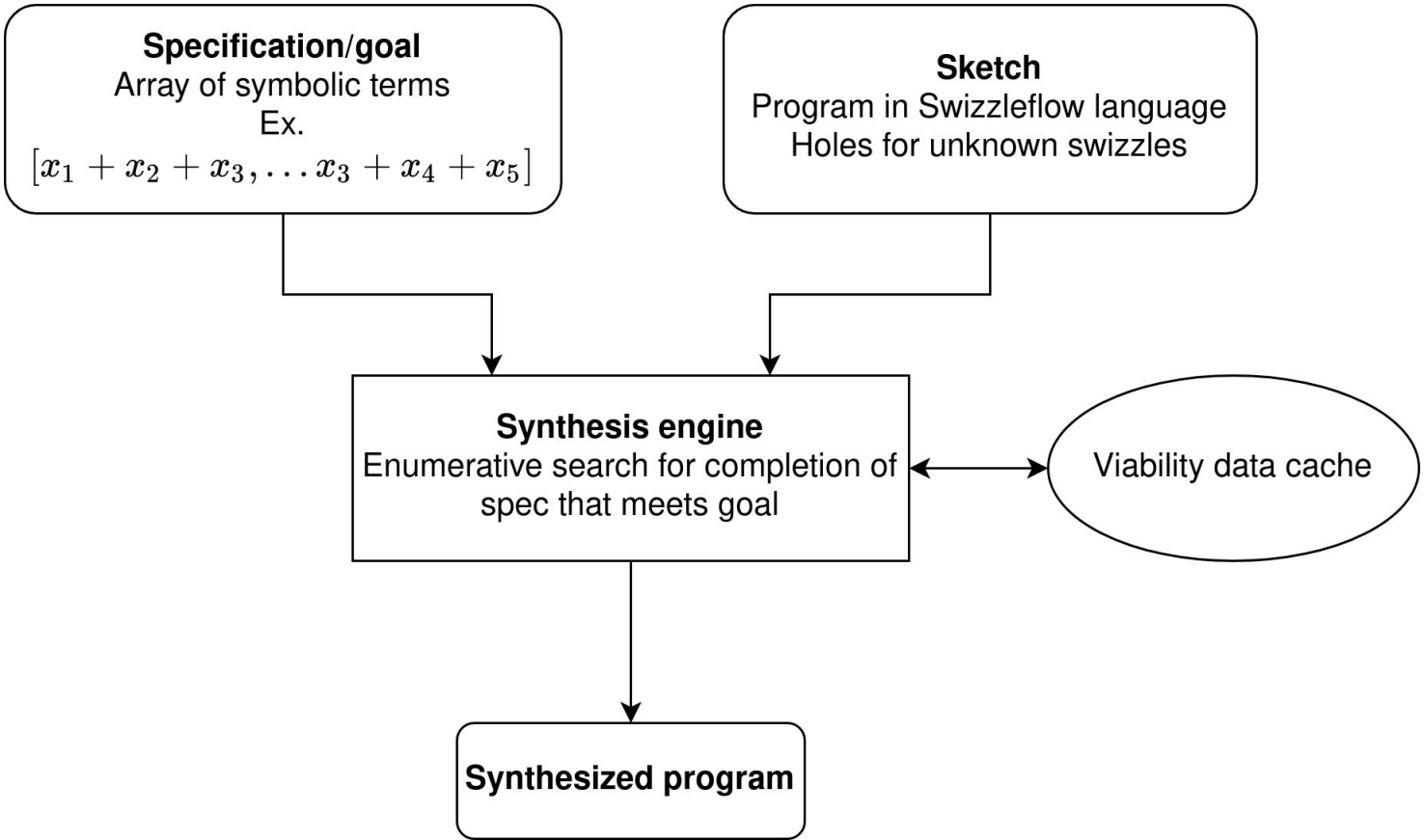



```
memory: [16]   = [x0, … x15]
input:  [4, 4] = reshape(memory)
s1: [4, 4] = swizzle_row(λi, j. j - i % 4)(input)
s2: [4, 4] = swizzle_col(λi, j. 3 * i + j % 4)(s1)
s3: [4, 4] = swizzle_row(λi, j. j + i % 4)(s2)
```

Swizzleflow code for fast GPU transpose algorithm, swizzles highlighted

Overview

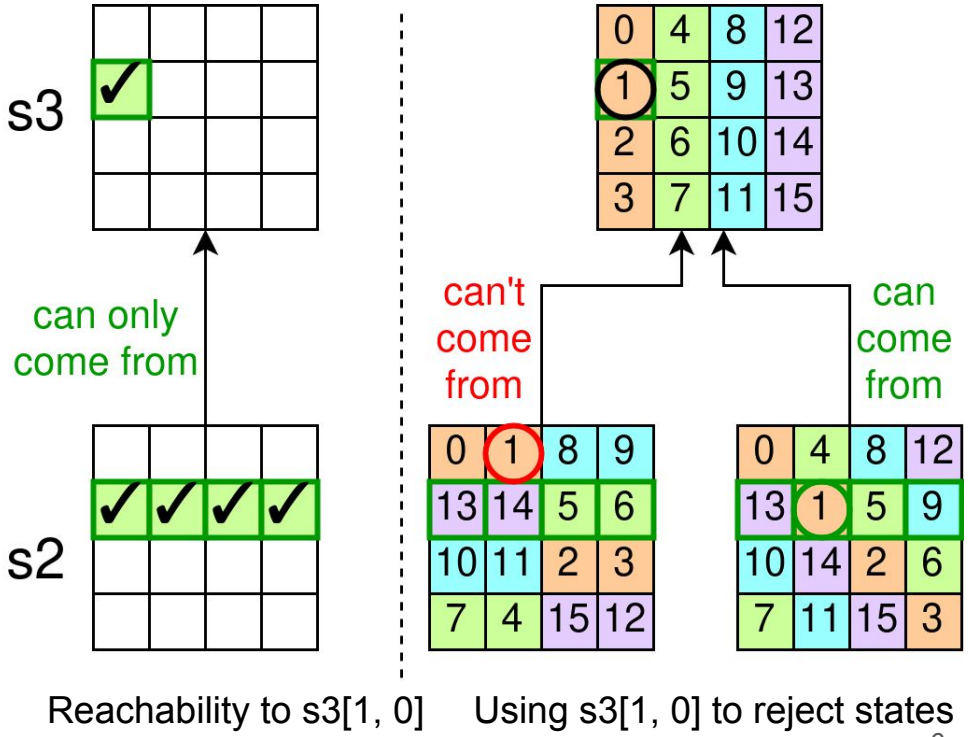
Synthesis



Viability tests

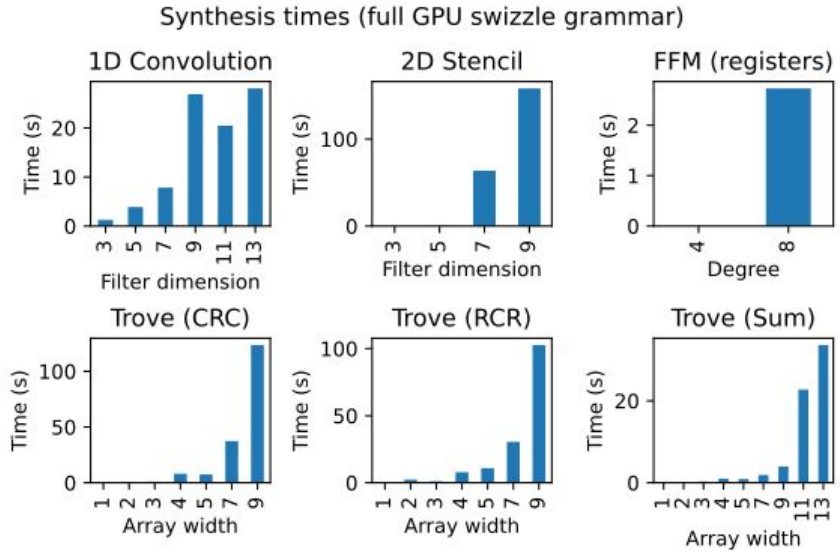
Our innovation: analyzing the synthesis problem means we can reject large parts of the tree without searching them

Ex. For each output location, is the value wanted there in a location it can come from given the rest of the program?



Preview of results

- Highly scalable algorithm
- Viability tests prune most of search tree (over 99%)
- Orders of magnitude improvements over previous work



Modeling swizzling kernels

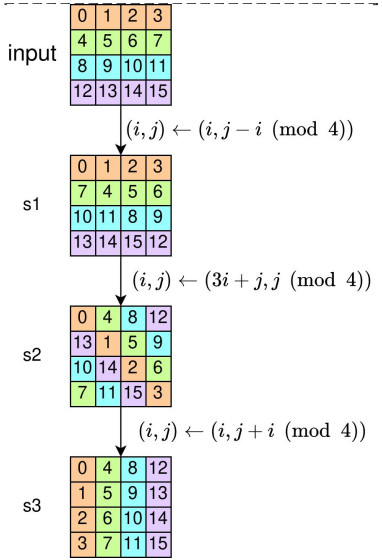
Efficient transpose on GPUs

```
for (int j = 0; j < T; j++)
  for (int i = 0; i < 4; i++)
    out[j][i] = input[i][j]
```

Specification

- T = 32, but 4 in examples
- 4 is smallest example of one solution class

```
int j = get_thread_id(); // T threads in parallel
register float s1[4], s2[4], s3[4];
for (int i = 0; i < 4; i++)
  s1[i] = input[i][j - i % T];
for (int i = 0; i < 4; i++)
  s2[i] = s1[(3 * i + j) % 4];
for (int i = 0; i < 4; i++)
  s3[i] = __shfl_sync(FULL_MASK, s2[i], j + i % T);
return s3;
```



Efficient swizzling implementation

Modeling swizzling kernels

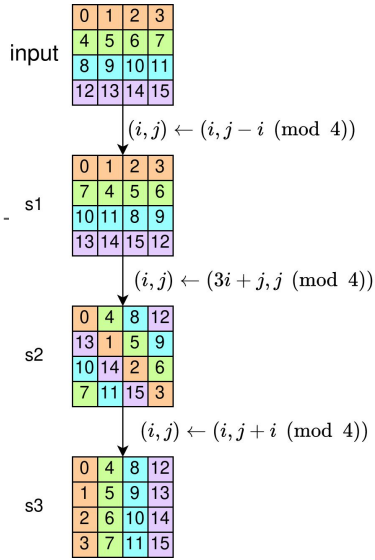
The core of swizzling transpose (1)

```
for (int i = 0; i < 4; i++)
  s1[i] = input[i][j - i % T];
for (int i = 0; i < 4; i++)
  s2[i] = s1[(3 * i + j) % 4];
for (int i = 0; i < 4; i++)
  s3[i] = __shfl_sync(FULL_MASK, s2[j], j + i % T);
```

GPU swizzling code

```
for (i in 0..4, j in 0..T)
  s1[i][j] = input[i][j - i % T];
for (i in 0..4, j in 0..T)
  s2[i][j] = s1[(3 * i + j) % 4][j];
for (i in 0..4, j in 0..T)
  s3[i][j] = s2[i][j + i % T];
```

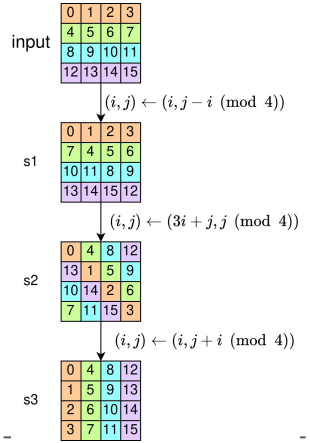
Core swizzles



Modeling swizzling kernels

The core of swizzling transpose (2)

```
for (i in 0..4, j in 0..T)
  s1[i][j] = input[i][j - i % T];
for (i in 0..4, j in 0..T)
  s2[i][j] = s1[(3 * i + j) % 4][j];
for (i in 0..4, j in 0..T)
  s3[i][j] = s2[i][j + i % T];
```

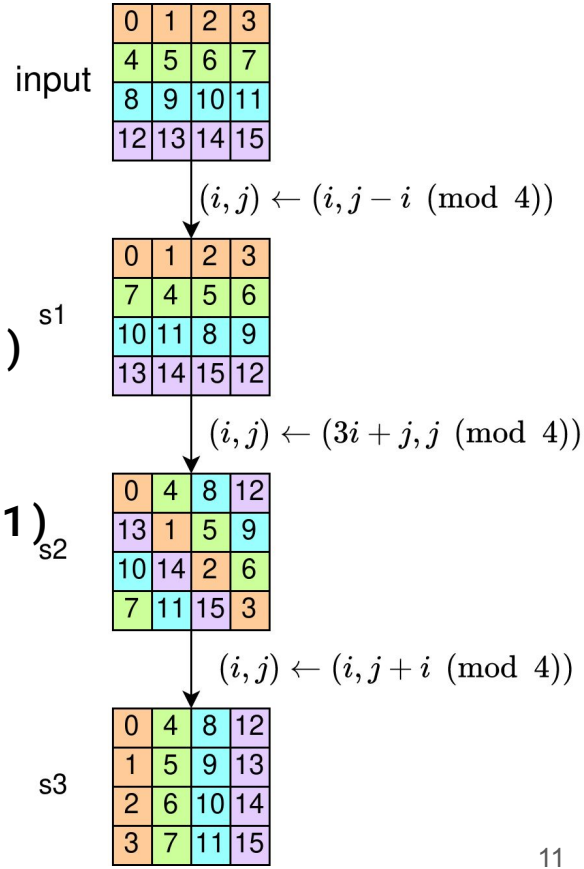



```
memory:  [16] = [x0, … x15]
input: [4, 4] = reshape(memory)
s1: [4, 4] = swizzle_row(λi, j. j - i % 4)(input)
s2: [4, 4] = swizzle_col(λi, j. 3 * i + j % 4)(s1)
s3: [4, 4] = swizzle_row(λi, j. j + i % 4)(s2)
```

Swizzleflow language gives a higher-level expression of the data movement

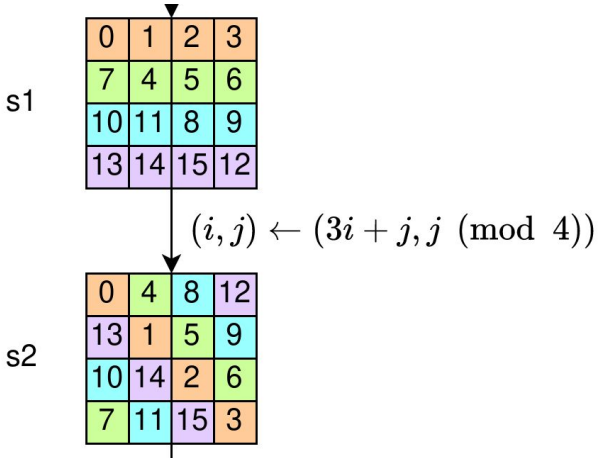
Drawing swizzle kernel executions

```
memory: [16]   = [x0, … x15]
input:  [4, 4] = reshape(memory)

s1: [4, 4] = swizzle_row(λi, j. j - i % 4)(input)

s2: [4, 4] = swizzle_col(λi, j. 3 * i + j % 4)(s1)

s3: [4, 4] = swizzle_row(λi, j. j + i % 4)(s2)
```



Gathers

- Each output element is either a load from an input, 0 (identity), or $\perp$ (undefined, shouldn't appear in output, etc.)
- No dependency on values in arguments
- General ex.: $f(x, y) = [x[0], y[1], x[0], \perp]$
- Model most aspects of a swizzle kernel
- Generalize permutations
  - Broadcast is a gather
- Inputs and output have fixed shape/type



Sketching transpose

```
for (int i = 0; i < 4; i++)
  s1[i] = input[i][?gpu_swizzle(j, i, 4, T)];
for (int i = 0; i < 4; i++)
  s2[i] = s1[?gpu_swizzle(i, j, T, 4)];
for (int i = 0; i < 4; i++)
  s3[i] = __shfl_sync(FULL_MASK, s2[i], ?gpu_swizzle(j, i, 4, T));
```

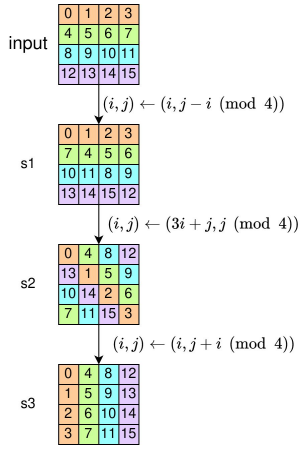
```
for (i in 0..4)
  for j in 0..T)
    assert s3[i][j] == input[j][i]
```

Imperative specification

Imperative sketch

```
for (int i = 0; i < 4; i++)
  s1[i] = input[i][j - i % T];
for (int i = 0; i < 4; i++)
  s2[i] = s1[(3 * i + j) % 4];
for (int i = 0; i < 4; i++)
  s3[i] = __shfl_sync(FULL_MASK, s2[i], j + i % T);
```

Imperative solution



Sketching transpose in Swizzleflow

```
memory: [16] = [x0, … x15]
input:  [4, 4] = reshape(input)
s1: [4, 4] = ?gpu_swizzle_row(input)
s2: [4, 4] = ?gpu_swizzle_col(s1)
s3: [4, 4] = ?gpu_swizzle_row(s2)
// specification
goal s3 == [[x0, x4, x8, x12], …]
```

Swizzleflow sketch

The holes (like `?gpu_swizzle_row`) are sets of gathers with the same type

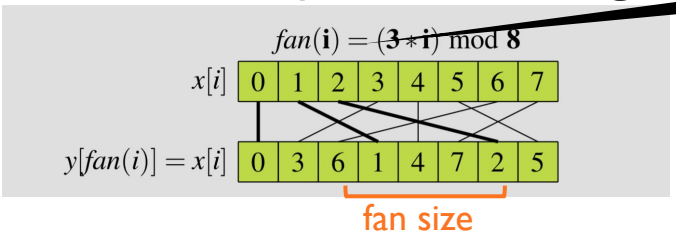
```
s1: [4, 4] = swizzle_row(λi, j. j - i % 4)(input)
s2: [4, 4] = swizzle_col(λi, j. 3 * i + j % 4)(s1)
s3: [4, 4] = swizzle_row(λi, j. j + i % 4)(s2)
```

Solution

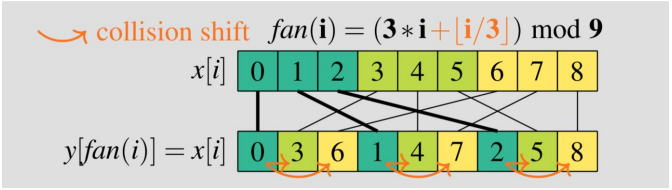
GPU swizzle template

?gpu_swizzle sketch defines the search space that contains **fanning** followed by **rotation** with optional grouping (taken from Swizzle Inventor) (Swizzleflow selects instructions on HVX

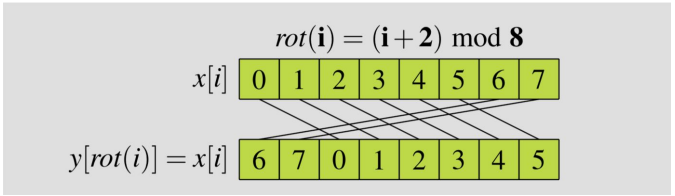
co-prime **fanning**



non-co-prime **fanning**



rotation



grouping

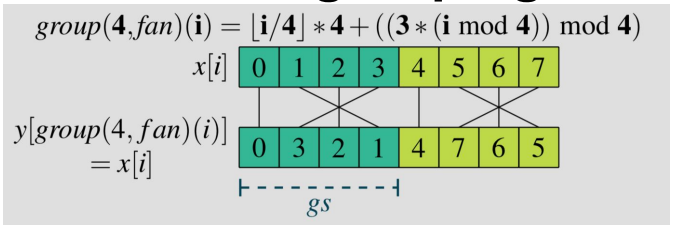
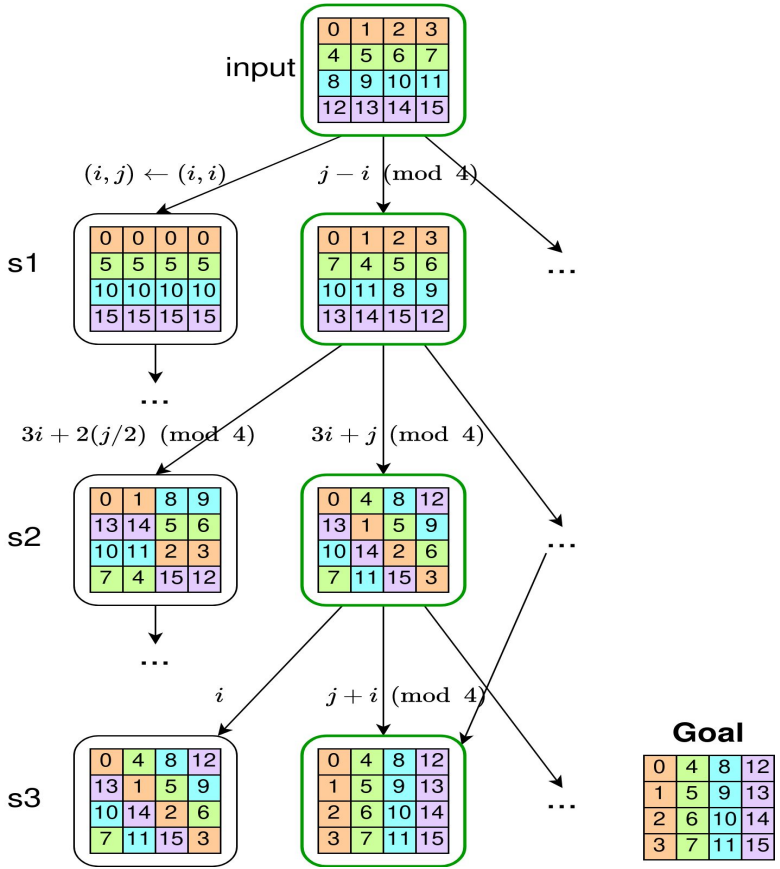


Figure from Swizzle Inventor (ASPLOS ‘19)

Synthesizing transpose in Swizzleflow

```
memory: [16] = [x0, … x15]
input:  [4, 4] = reshape(input)
s1: [4, 4] = ?gpu_swizzle_row(input)
s2: [4, 4] = ?gpu_swizzle_col(s1)
s3: [4, 4] = ?gpu_swizzle_row(s2)
// specification
goal s3 == [[x0, x4, x8, x12], …]
```

Solve with enumerative search



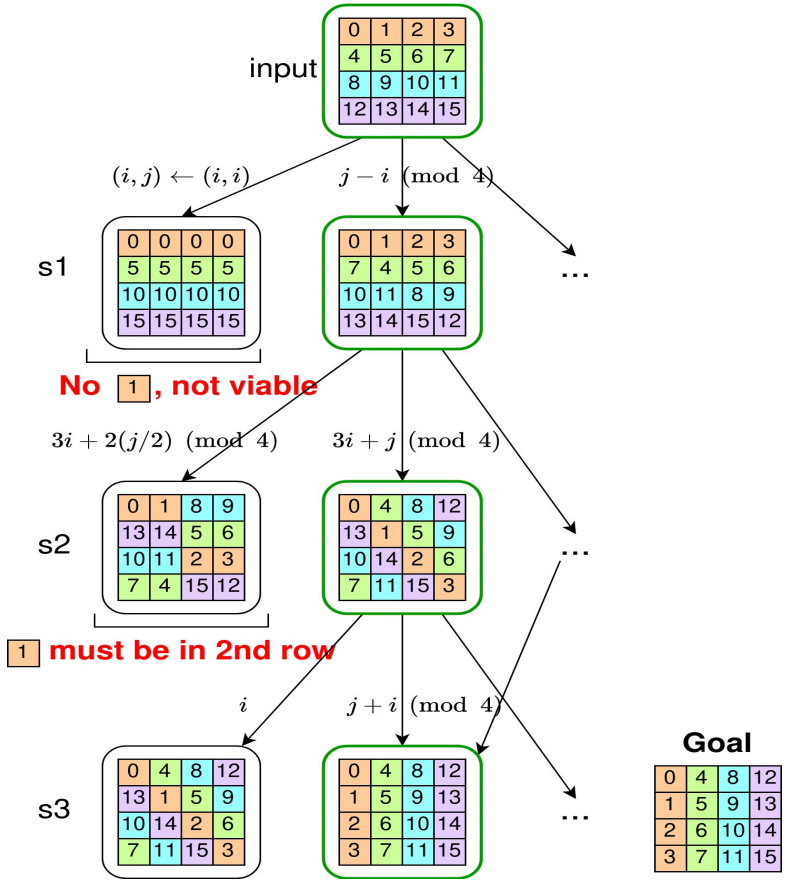
Viability tests

Non-viable intermediate states

~10^{13} paths for 32 by 4 transpose!

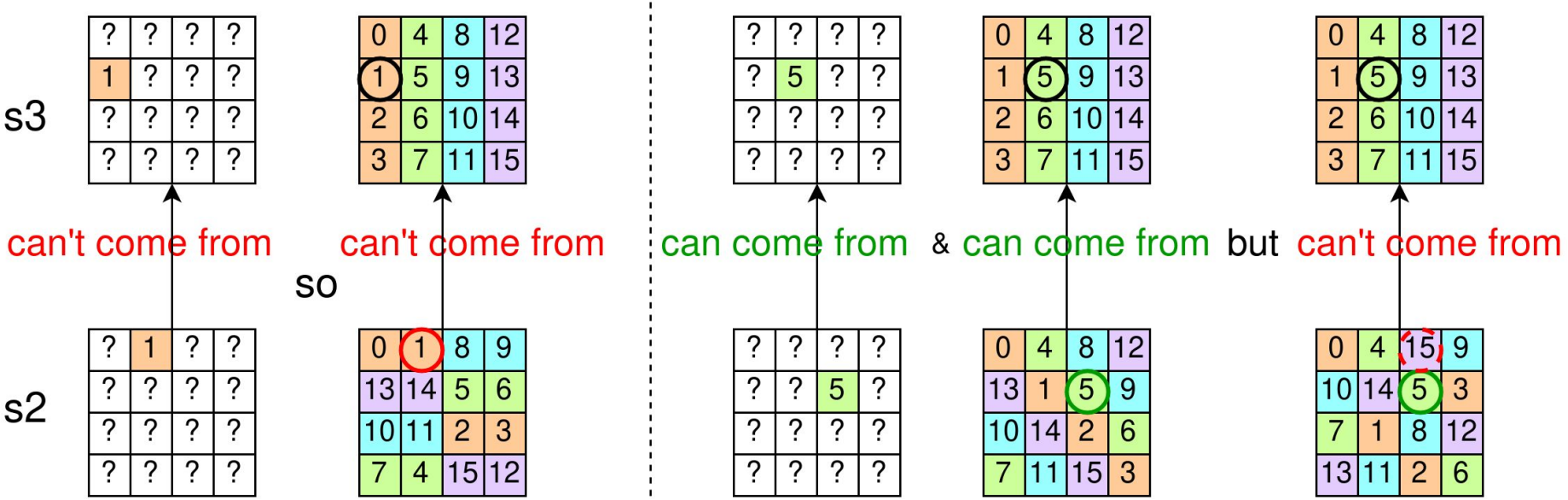
Can't explore them all

How do we detect the non-viable ones?



Viability tests

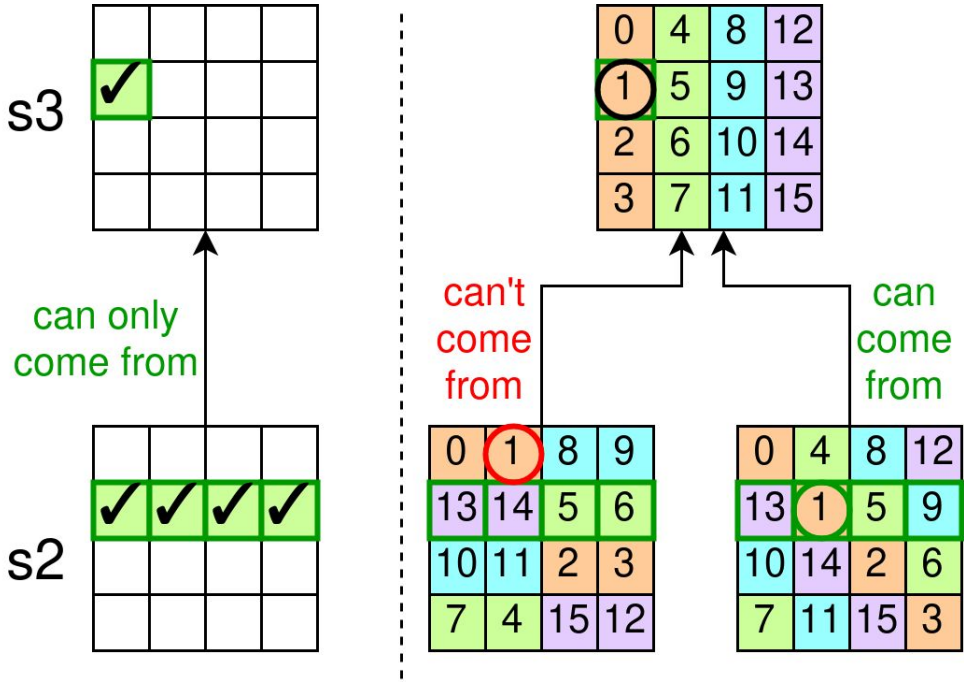
The reachability property



Each output location's value must be able to come from a location in the current state with that value.

Viability tests

Pre-computing reachability

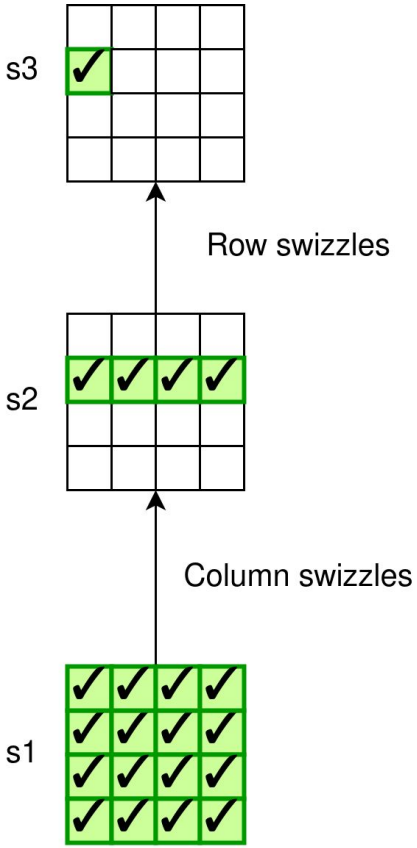


Reachability to s3[1, 0]

Using s3[1, 0] to reject states

- Find source locations in candidate that output location can come from
- Check if a location each output can come from contains needed value
- If any fails, reject state

Simultaneously tracking multiple locations



- Tracking one location isn't enough
  - Quickly find that every location is a source
  - Causes benchmarks to time out
- "Where can outputs *i* and *j* come from together?"
- Two locations is sufficient
  - 99.6% of states skipped on average
  - 45% of benchmarks with solutions prune optimally
- 3+ locations
  - Data is too big (~1M nodes per output triple for 32x4) — timeout when computing

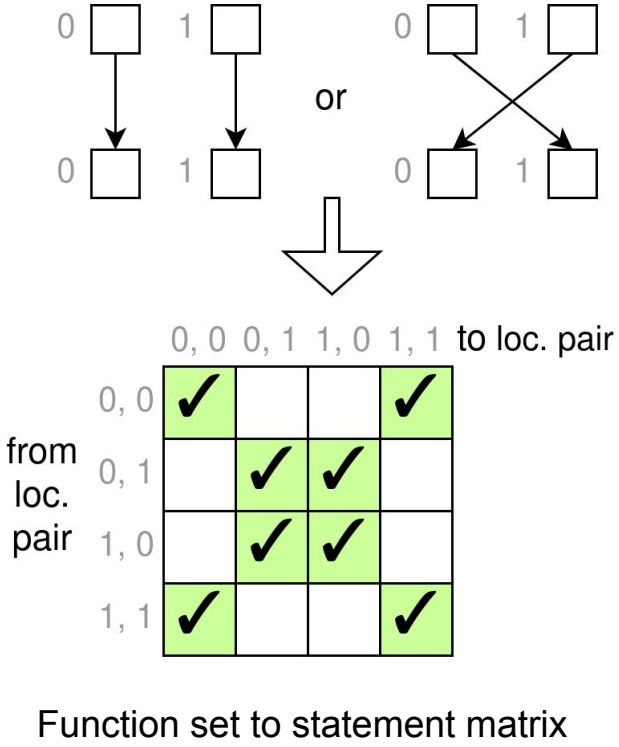
Reachability testing is dataflow analysis

- “Which pairs of locations can the values in this pair of output locations have come from?” is a dataflow analysis
- Characteristics
  - Finite (fixed set of locations we track)
  - Distributive (only need to look at one function output to know where it can come from so the analysis distributes over union)
  - Subset (we care about which locations can reach)
- Can be converted to graph reachability[1]
  - Each statement“explodes” into one node per tracked location

[1] Reps, T., Horwitz, S., & Sagiv, M. (2003). *Precise interprocedural dataflow analysis via graph reachability*.. POPL ‘95

Computing reachability

1. For each (set of) functions, compute possible dataflow from outputs to inputs
2. Compose resulting statement matrices with multiplication
3. This solves reachability problem for all output location pairs *at once*
   a. All dataflow graphs have the same structure, only the *x* in “The values in *x* can com from where?” changes



Function set to statement matrix

Reachability matrix statistics

- All but two statement matrices for GPUs have < 5% bits set
  - 82% of them have less than 1% of entries set
- Compositions (row + column swizzles) raise density, still works
- Exploiting sparsity and boolean multiply took 32x5 transpose multiply times to 0.08 to 7 seconds each from 120s each (dense floats)

Modeling swizzling kernels

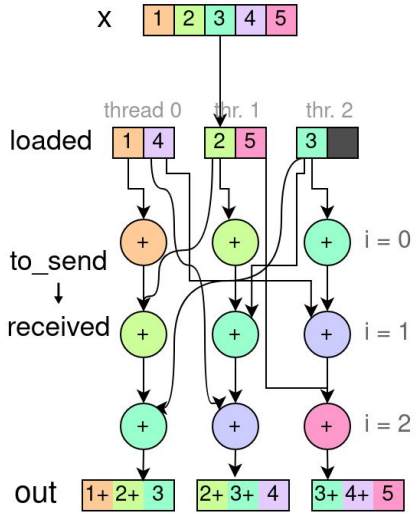
Convolution

```
// Spec
for (int i = 0; i < W; i++)
  out[i] = 0
  for (int j = 0; j < K; j++)
    out[i] += x[i + j]
```

Imperative convolution program

Threads load values to registers and read from their neighbors

```
// on W (generally 32) threads
int i = get_thread_id();
float loaded[2] = {x[i], x[W + i]};
float accum = 0;
for (int j = 0; j < K; j++)
  float to_send = loaded[i >= j ? 0 : 1];
  float received = __shfl_sync(FULL_MASK,
                               to_send, i + j % W);
  accum += received;
out[t] = accum;
```



Modeling swizzling kernels

Convolution

```
// on W (generally 32) threads
int i = get_thread_id();
float loaded[2] = {x[t], x[W + t]};
float accum = 0;
for (int j = 0; j < K; j++)
   float to_send = loaded[i >= j ? 0 : 1];
   float received = __shfl_sync(FULL_MASK, to_send, i + j % W);
   accum += received;
out[t] = accum;
```

Imperative convolution program

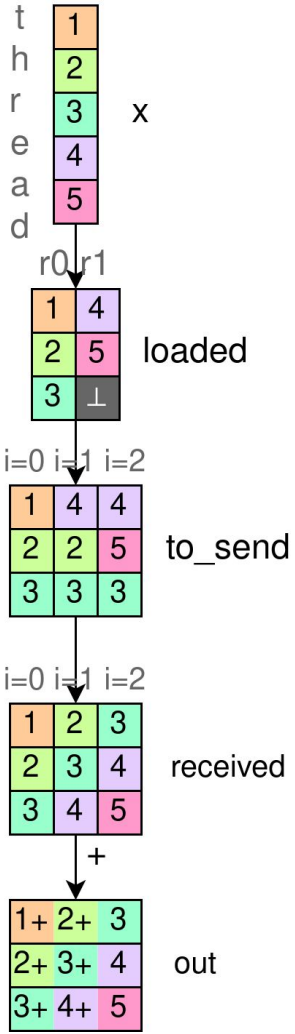
```
x: [34]          = [x0, ... , x34]
// arrays are threads by registers/iteration
// put ⊥ in out-of-bounds locations
loaded:   [32, 2] = load_trunc(x)
to_send:  [32, 3] = select(λi, j. i >= j)(loaded)
received: [32, 3] = swizzle_col(λi, j. i + j % 3)(to_send)
out:      [32]    = fold(+) received
```

Swizzleflow convolution program

Loop unrolled, arithmetic abstracted

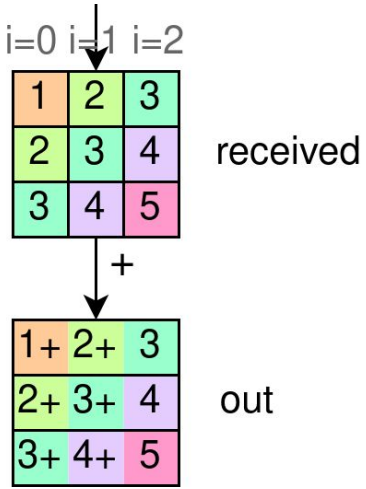
Convolution execution

```
x: [5]           = [x1, ... , x5]

loaded: [3, 2]   = load_trunc(x)

to_send: [3, 3]  = select(λi, j.i >= j)(loaded)

received: [3, 3] = swizzle_col(λi, j. i + j % 3)(to_send)

out: [3]         = fold(+) received
```



Folds

- Represent reductions (+, *, max, …)
- Order of elements doesn't matter
- $$\text{fold}\left(\begin{bmatrix} a & a \\ d & c \end{bmatrix}\right) = \begin{bmatrix} \text{fold}\{a, a\} \\ \text{fold}\{c, d\} \end{bmatrix}$$
- $$\text{fold}\{a_1, \ldots, a_k, 0\} = \text{fold}\{a_1, \ldots, a_k\}$$
- fold{} = 0
- $\text{fold}_*\{\text{fold}_+\{a, b\}, c\}$ is (a + b) * c
  - Not equal to $\text{fold}_+\{a, b, c\}$

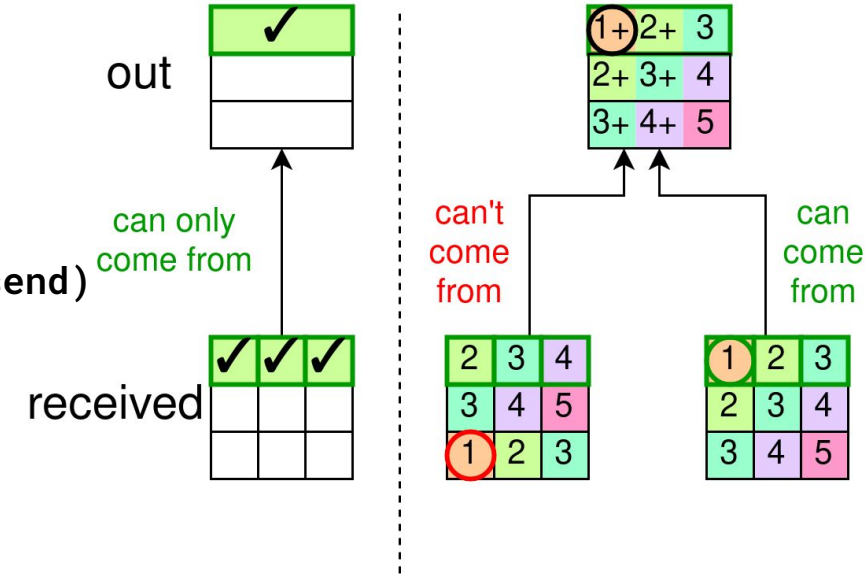


Gather and fold are enough

- Fixed-size arrays: kernels have bounded inputs, loops
- No `if (x[i] < 0)` — simplifies analysis
- No loop-carried dependencies except accumulation
- Limitations
  - No data-dependent indexing → no sparsity (which is hard)
  - No lookup tables (ex. Inverse sqrt() on HVX)
  - HVX: No built-in way to find arithmetic instructions

Convolution synthesis

```
x: [34] = [x0, ... , x34]

loaded: [32, 2]   = load_trunc(x)
to_send: [32, 3]  = ?select(loaded)
received: [32, 3] = ?gpu_swizzle_col(to_send)

out: [32] = fold received

// [x0 + x1 + x2, x1 + x2 + x3, …]
goal out == conv{k=3}([x0, ..., x34])
```

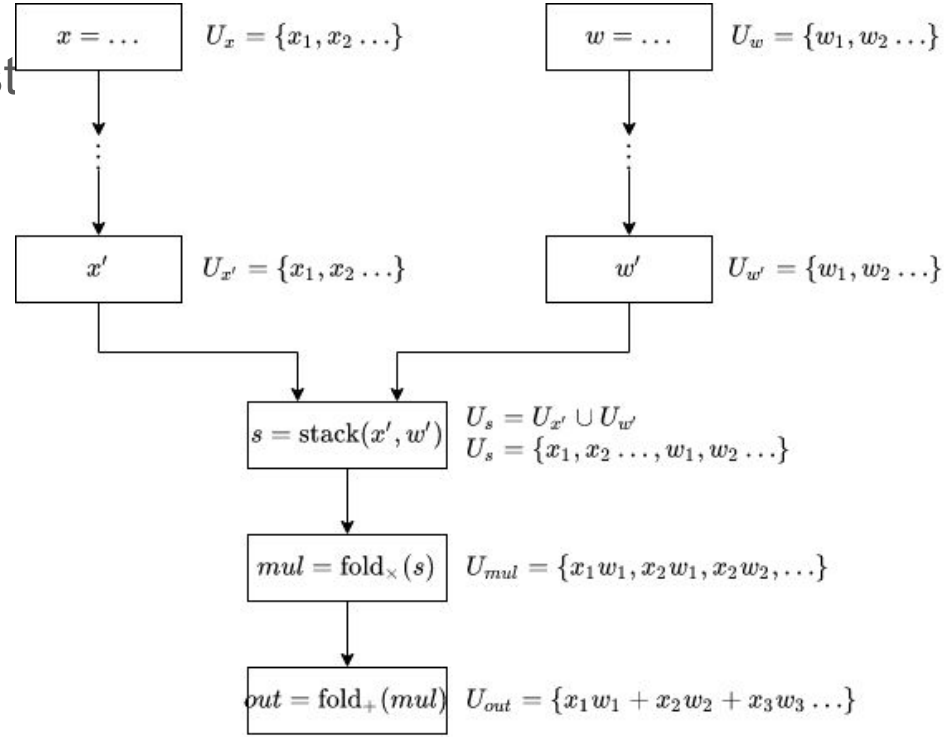


Problem: We want to see where 1+ 2+ 3 can come from, but only have 1, 2, and 3 available earlier in the search.

Solution: Ensure each available *subterm* (pair) reaches the output. For analysis, folds put multiple values in a box

Universes

- Question: Which terms could exist in in a given variable?
- Relevant to
  - Folds (this is more precise than using symbols)
  - Multiple inputs (see right)
- Each variable gets a universe
- Use subterms in universes of all live variables



Copy counts

For the sketch

```
x = ?pick(a, 0)
y = ?pick(a, 0)
z = ?pick(b, 0)
o = x + y + z
assert o == a + b
```

the partial program

```
x = a
y = a
z = ?pick(b, 0)
o = x + y + z
assert o == a + b
```

cannot be correct. There are too many copies of a.

The reachability test won't detect this.

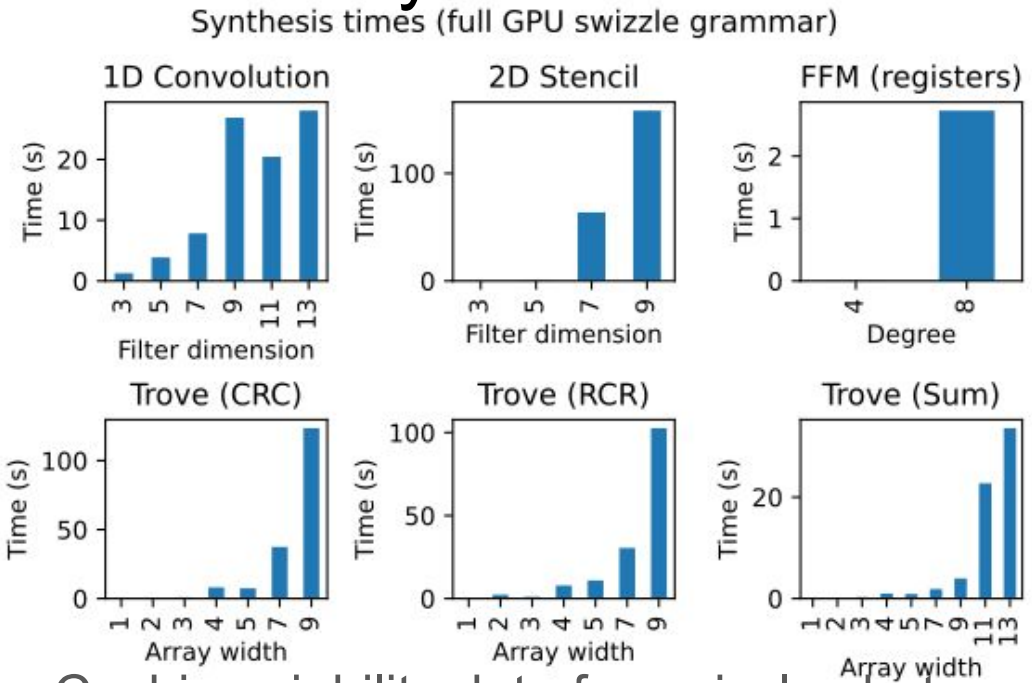
- Comes up in polynomial multiply — 16 `keep_if()`s, 8 are false
- Detecting extra term copies brings that search from timeout to 3s
- Compute bounds on how often the value in each location appears in the whole output. Reject if sum of bounds for locations containing a term doesn't include true count.

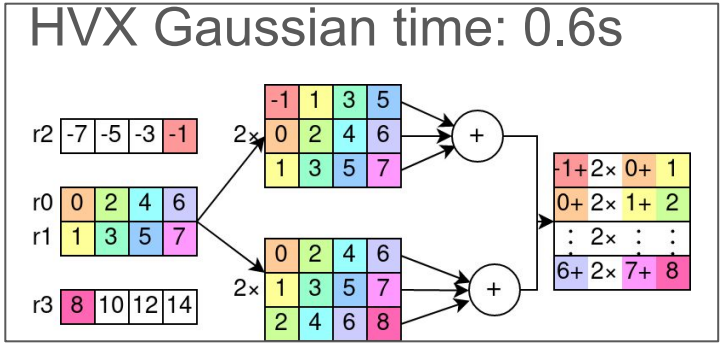
Evaluation questions

1. Scalability with the problem size
2. Efficiency of pruning
3. Comparison with Swizzle Inventor

Results

Q1: Scalability





- Caching viability data for swizzles between benchmarks is helpful
- Trove is 32 x width transpose
- Searching for all solutions: determinism & no cost model

Q1; Scalability limits

- Running out of memory
  - 32x11 transpose (each matrix has $(32 * 11)^4$ bits)
  - 15-long 1D convolution
  - 32x15 transposed sum
- Function set creation time (usually negligible)
  - 11x11 2D stencil (needs *cond1* : 0 ? *cond2* : 1 ? *cond2* ? 2 : 3)
- Abstraction limits
  - Width-8 polynomial multiply with shared memory needs an hour
    - Only Swizzle Inventor benchmark we can’t do

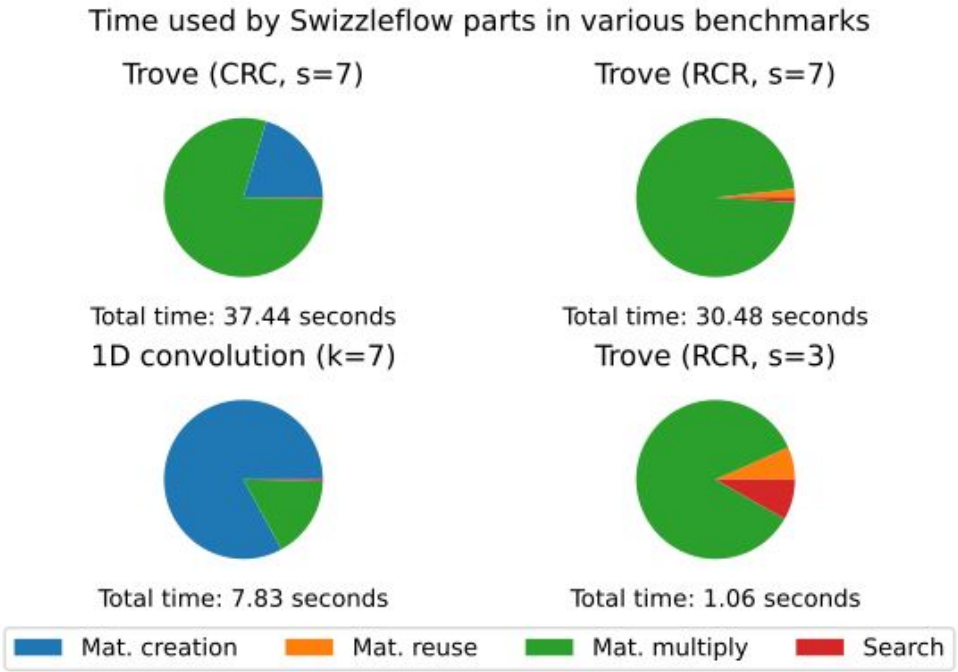
Results

Q2: Pruning effectiveness

Benchmark	States visited / oracle	States visited	Search space size
2D stencil (k=5)	1.00	113	4.4e07
2D stencil (k=7)	1.01	4598	1.4e11
Trove, CRC (s=7)	1.01	3534	4.6e11
Trove, RCR (s=7)	2.24	14363	2.2e13
1D convolution (k=3)	1.93	2894	1.9e09
FFM (width=8, registers	27.6	67472	2.6e42
HVX Gaussian instructions	51.3	116111	1.5e15

45% of menchmarks had oracle-like performance

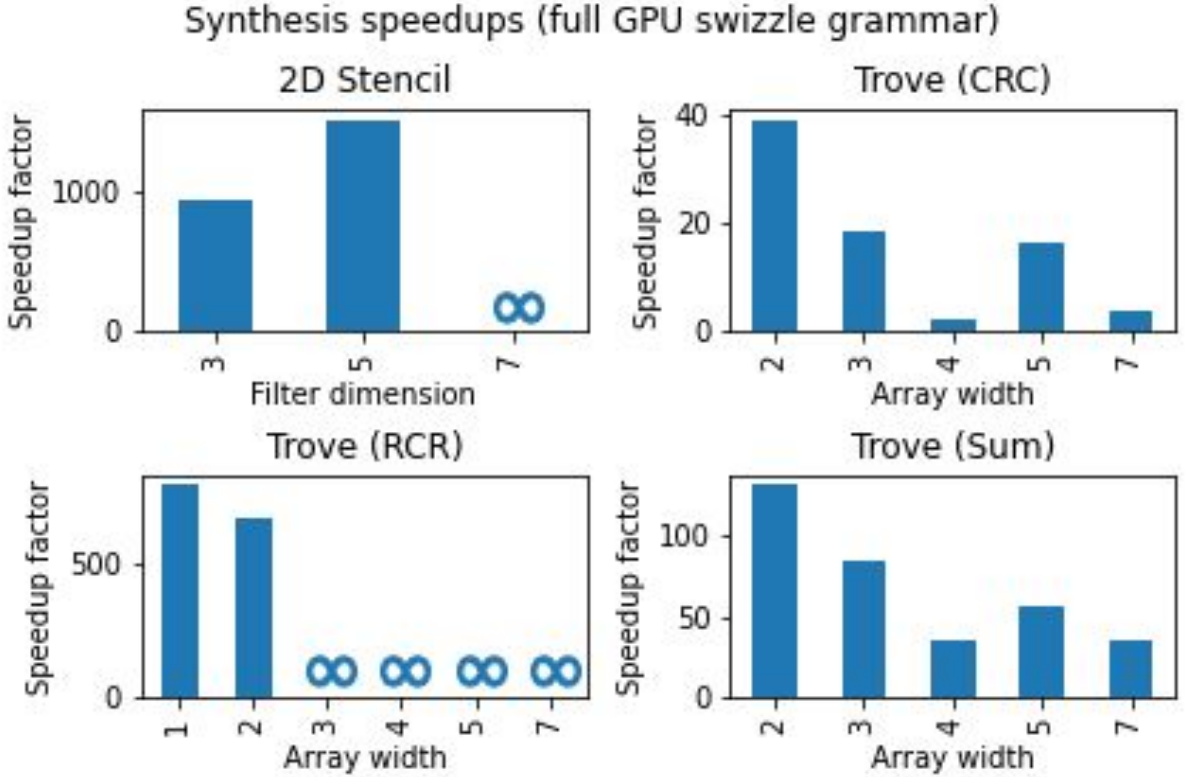
Q2: Effect of abstraction on search



- Search almost always <1s
- Matrices are cached between problems
- Highly effective pruning reduces search time
- Boolean matrix multiply

Results

Q3: Speedups over Swizzle Inventor



Q3: Removing grammar restrictions

- For larger benchmarks, Swizzle Inventor used GPU swizzle subset
- In Swizzleflow, we didn’t need this (ex. could add back grouping)
- Pre-computing pruning faster than ad-hoc SMT discovery
  - Except for CRC Trove with restricted grammar and shmem FFM
- Using sets of functions allowed semantic deduplication of grammar

Related work

- Dataflow-based pruning for superoptimization (OOPSLA ‘20)
  - Prunes search for best optimization of LLVM IR
  - Reasons about what bits can/must be set to reject holes
  - Ex. `x << C` can’t optimize `(x + 1) | 1`

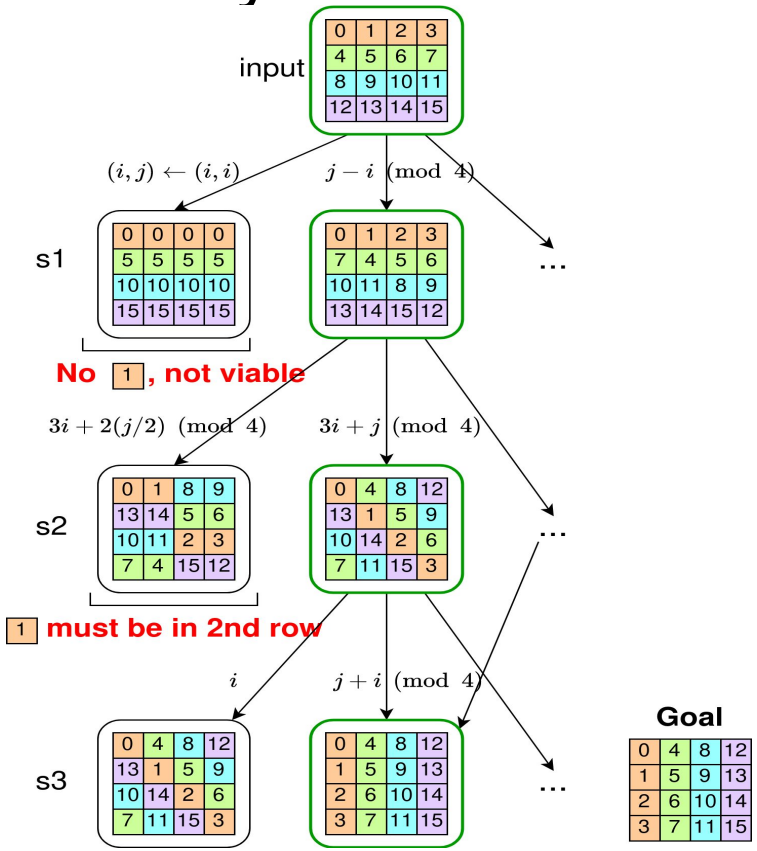
Acknowledgements

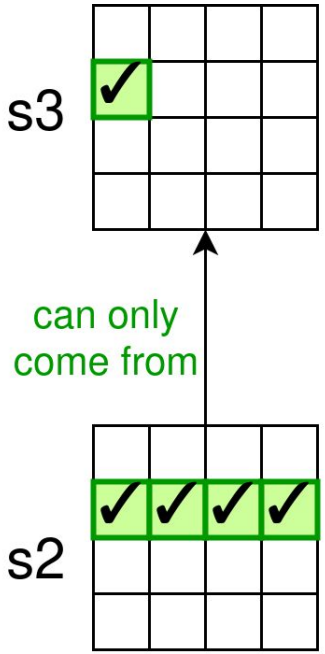
- Ras Bodik —advice, feedback, pushing to simplify the presentation
- Sam Kaufman — Helpful sounding board when I was stuck
- Mazz Ahmad — HVX challenge problems and performance numbers
- An Wang — Exploring transpose solutions to give benchmarks

Summary

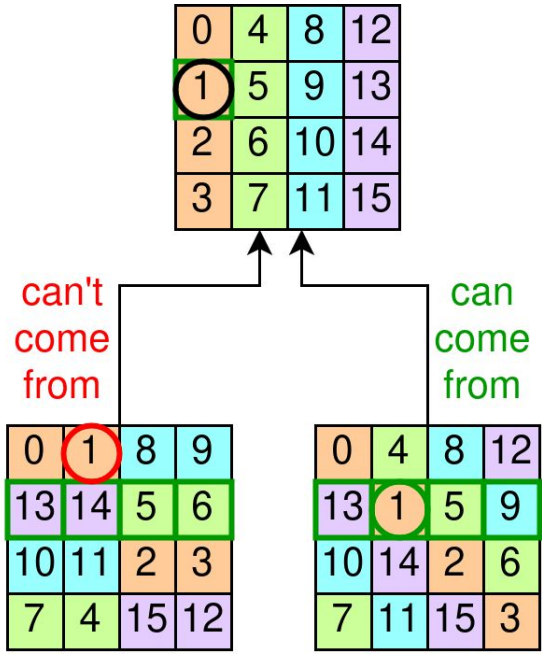
1. Accelerators need swizzles for performance
2. Swizzleflow models swizzling programs on multiple platforms.
3. Swizzleflow allows programmers to synthesize swizzles.
4. New viability tests allow us to prune most of our search space.
5. We are more scalable than previous work because of this.

Thank you all for coming! Questions?





Reachability to s3[1, 0]



Using s3[1, 0] to reject states

The slide graveyard

Why synthesis and not a compiler?

- Compilers change a program locally to optimize
- Changing whole data movement isn't made of optimization steps
- Need search and/or human input

```
for (int j = 0; j < T; j++)
  for (int i = 0; i < 4; i++)
    out[j][i] = input[i][j]
```

```
int j = get_thread_id(); // T threads in parallel
register float s1[4], s2[4], s3[4];
for (int i = 0; i < 4; i++)
  s1[i] = input[i][j - i % T];
for (int i = 0; i < 4; i++)
  s2[i] = s1[(3 * i + j) % 4];
for (int i = 0; i < 4; i++)
  s3[i] = __shfl_sync(FULL_MASK, s2[i], j + i % T);
return s3;
```

Why Swizzleflow?

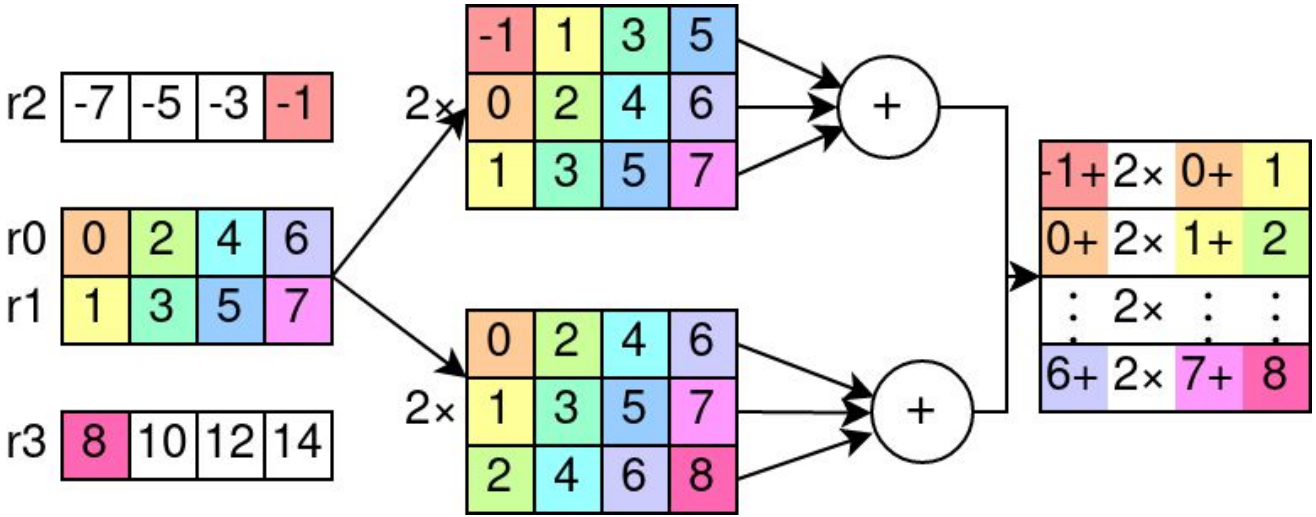
- Other codegen tools don’t focus on swizzles
  - AutoTVM, Lift, etc. work on scheduling - breaking a large problem
    - Assume optimized kernels
    - Swizzleflow could be integrated here
- Address limitations of other synthesis tools
  - Much work focuses on one platform (SIMD, GPU)
  - Swizzle Inventor (previous work) only targets GPUs and significantly limited search space for efficiency
    - We resolve these limitations

[1] Phitchaya Mangpo Phothilimthana et al.. Swizzle Inventor:Data Movement Synthesis for GPU Kernels. 2019. In ASPLOS.

Accelerator programming is hard

- Accelerators include GPUs, vector processors (HVX, SIMD), FPGAs
- Used for fast math in multiple domains
- To be fast, accelerators constrain programs & offer special features
- Ex. GPUs want memory read in blocks and have inter-thread shuffle
- S*wizzles*: irregular mappings of data to memory and compute
  - Include things like modulo — not just tiling or affine
  - If not made by hand, tools usually only look for affine maps

Accelerator constraints: HVX Gaussian



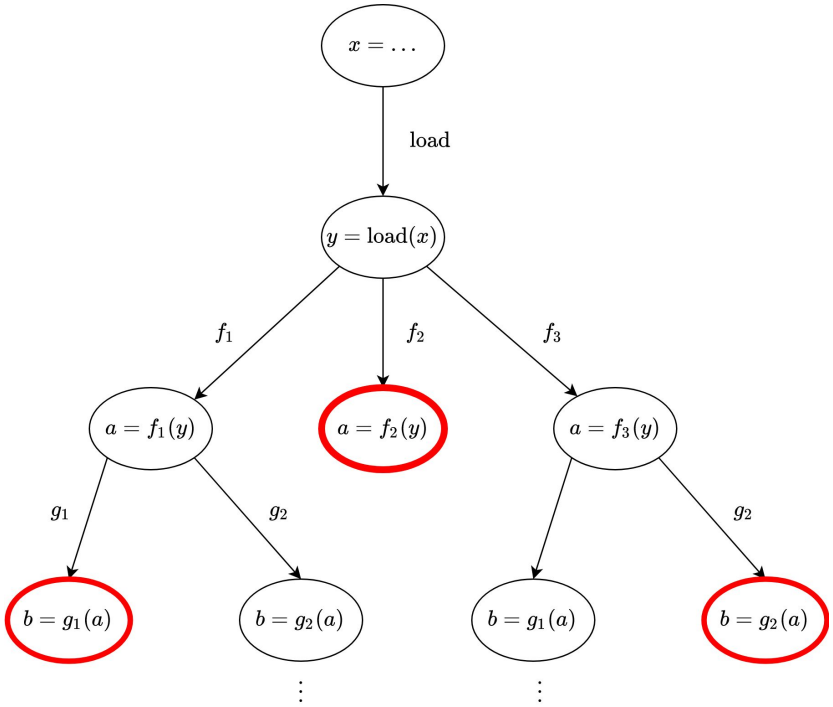
- Many HVX instructions work on odd-even register pairs
- Exploiting this layout gives 2x speedup over Halide
  - Halide inserts a lot of unpacking and repacking

The Swizzleflow language - partial grammar

```
Program ::= Statement+
Types are the dimensions of each array
Type ::= '[' (integer ',')* integer ','? ']'
Goals and function definitions omitted
Statement ::= 'goal' ':' Type GoalDef | 'define' FunctionsDef
            | variable ':' Type '=' Operation
Operation ::= 'fold' variable Abstract arithmetic
            A gather, or set of them, with optional arithmetic after
            | 'fold'? '?'? func_name '(' (variable ',')* variable ')'
            | Literal
```

Search

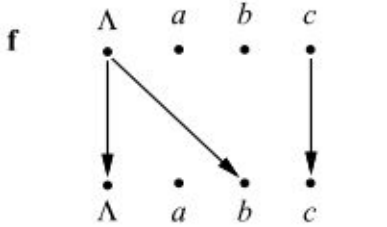
- At each statement, choose value for hole
- If current state is viable, search next statement
- On non-viable or wrong result, backtrack
- Can parallelize independent initial segments
- Works well due to viability tests
- Get all results because no cost model



IFDS - backup

- Interprocedural Flow-Sensitive Dataflow Analylis by Reps[1]
- (We don’t need procedures)
- Can convert dataflow into graph reachability
- Domain of dataflow facts *D*
- Each program edge has f : $2^D \rightarrow 2^D$ that distributes over union
  - Encodes transitions between facts
- “Explode” program graph to track reachability between dataflow facts

$$\mathbf{f} =_{df} \lambda S.(S - \{a\}) \cup \{b\}$$



Reps, T., Horwitz, S., & Sagiv, M. (2003). Precise interprocedural dataflow analysis via graph reachability.

Contents of Swizzleflow arrays

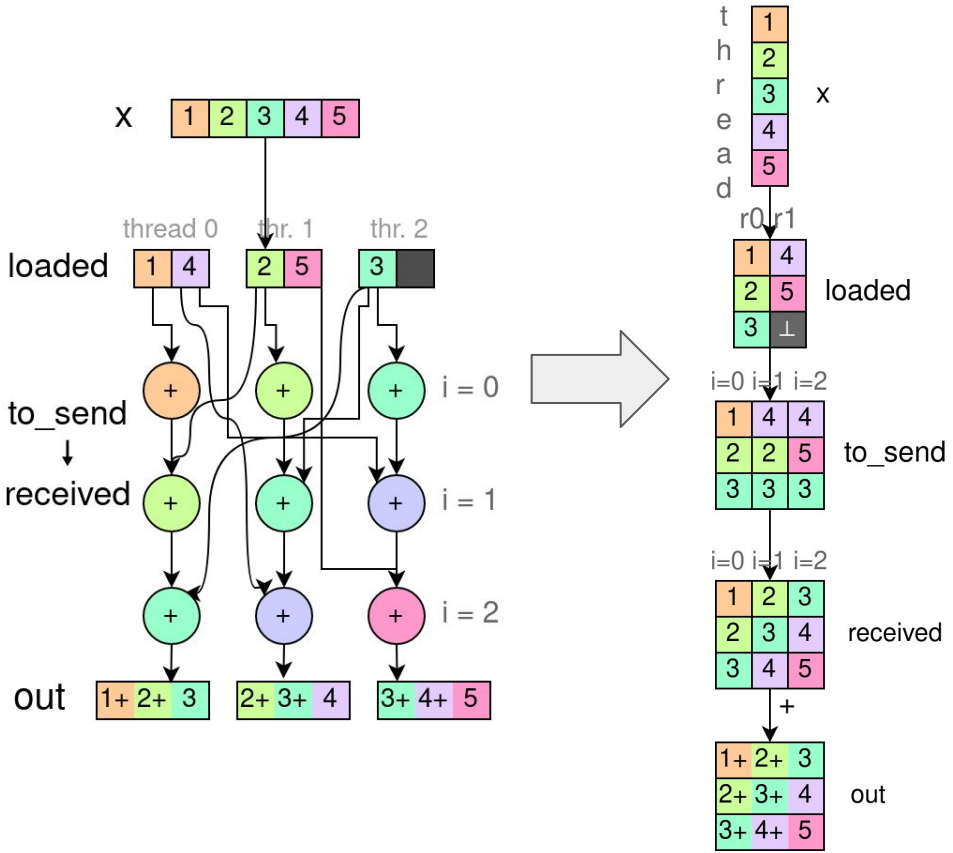
```
Value ::= Identity 0 |  Not defined ⊥ | Term
Symbol ::= Defined by problem [word not otherwise reserved, ex. x1]
Term ::= Symbol | ‘fold’ ‘{‘ (Term ‘,’)* Term ‘}’
```

- 0 is the identity
- ⊥ includes out-of-bounds read, but also any term that isn’t a subterm of the goal (as an optimization)
- fold{a_1, …, a_k, 0} = fold{a_1, … a_k}
- fold {} = 0
- fold{fold{a, b}, c} is , for example, (a + b) * c and is *not* fold{a, b, c}

Code for sketch

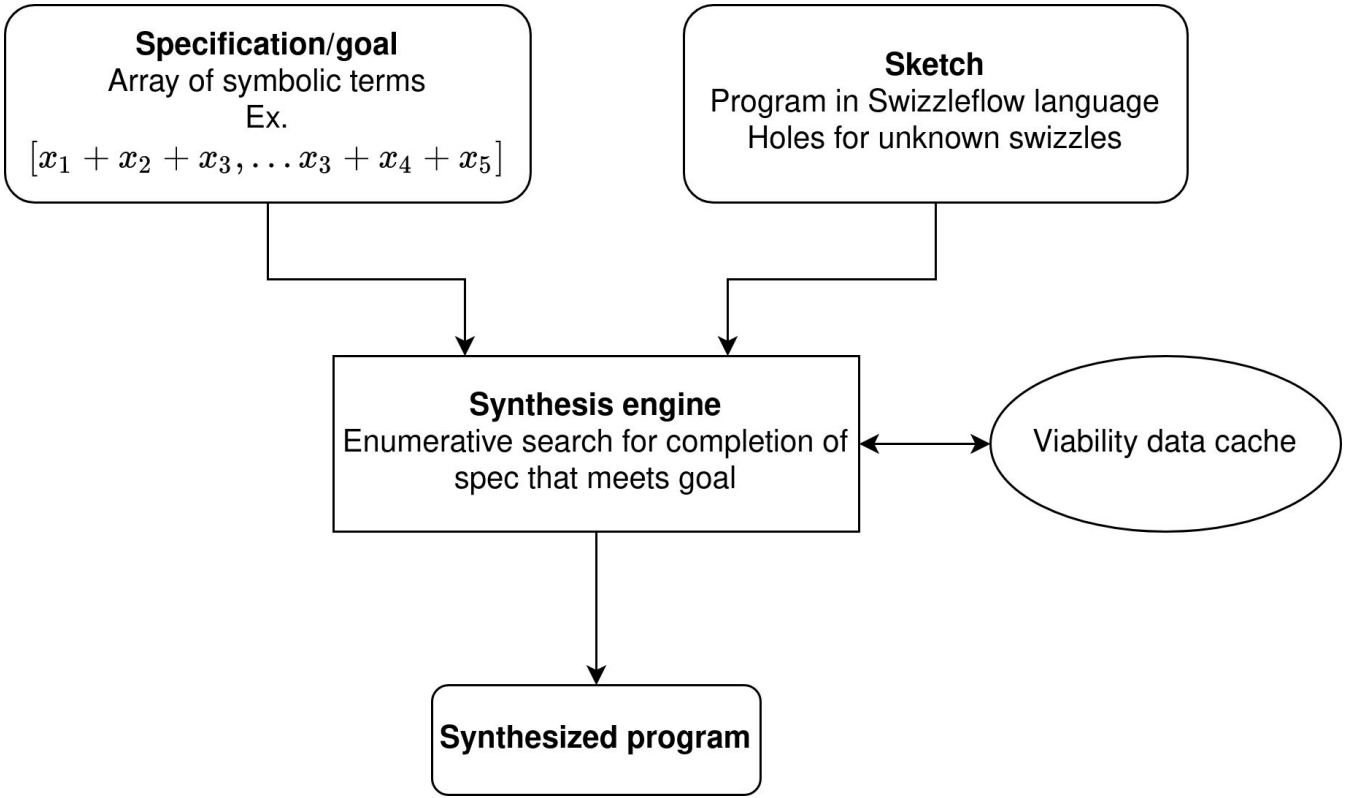
```
x: [34] = [x0, ... , x34]

loaded: [32, 2] = load_trunc(x)
to_send: [32, 3] = ?reg_select(loaded)
received: [32, 3] = ?col_swizzle(to_send)

out: [32] = fold
    ?cond_keep{fold_len=3}(received)

goal: [32] conv{k=3}([x0, ..., x34])
```

The Swizzleflow language



- Portable model of swizzling kernels as mapping between arrays
- Focuses on data movement
- Threads, loops, etc. unrelled array dimensions
- Arithmetic → folds
- Restrictions come from doma — make reasoning easier

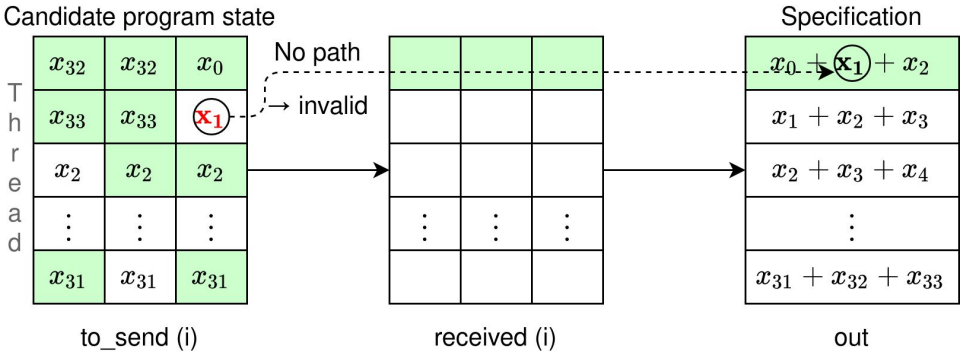
Synthesis problem



All gathers in each hole have the same type

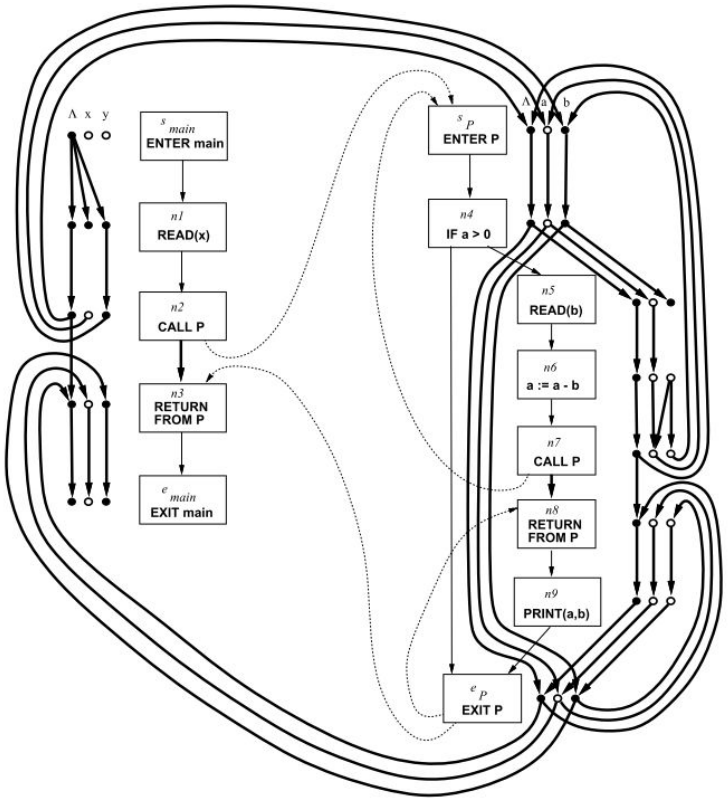
Viability tests

- Approximate which executions can be correct
- Use this to prove states can't be part of a correct program
- Main test uses a dataflow analysis to reject unworkable candidates
- Notation: v^i is the variable that holds v[i]



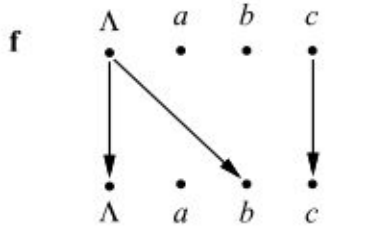
Viability

IFDS



Maybe-uninitialized variables example for IFDS

$$\mathbf{f} =_{df} \lambda S.(S - \{a\}) \cup \{b\}$$



- Convert dataflow analysis to graph reachability
- We don't need interprocedural
- "Explode" graph to reachability between dataflow facts
- Functions distribute over union/branches

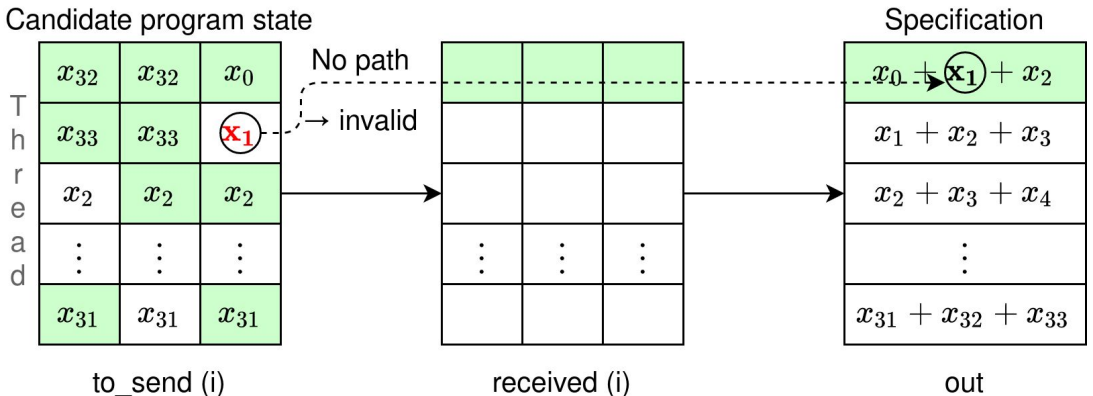
Reps, T., Horwitz, S., & Sagiv, M. (2003). Precise interprocedural dataflow analysis via graph reachability.

Dataflow in Swizzleflow

- We have a family of dataflow problems
- Effectively like a taint analysis
- Reaches_{g^i} - at each statement, the set of v^j such that the value in v^j can be placed in g^i where g is the program output/goal
- Works backwards from the goal
- Can compute by looking at each other, taking unions, composing
- Reality: we need Reaches_{g^i, g^j} for pairs of locations
  - Tracking one location means rotations can do anything
- Also abstracts folds — multiple values reach one output
- **TODO: draw this**

The pair-reachability test

- Each (pair of) locations in the goal must be reachable by correct terms, or candidate must be wrong
- For each *a, b* in universe(stmt)^2 and each g^i, g^j such that $a \in g[i]$ and $b \in g[j]$
  - Must have v^l, w^m in Reaches_{g^i, g^j}(*stmt*) so *v[l]* = a, *w[m]* = b



Universes

- Which terms to test?
- Ex. goal has a + b, but only have a and b
- Or - literal for constants not run yet
- Solution: each variable (and statement, through union over live vars) gets a universe of potential terms
- How?
  - Literal: all terms in the literal that are subterms of the goal
  - Gather: union of universes of arguments
  - Fold: all combinations of universe of argument that are in goal
- **TODO: add example/draw one/clean up these bullet points**

The global graph

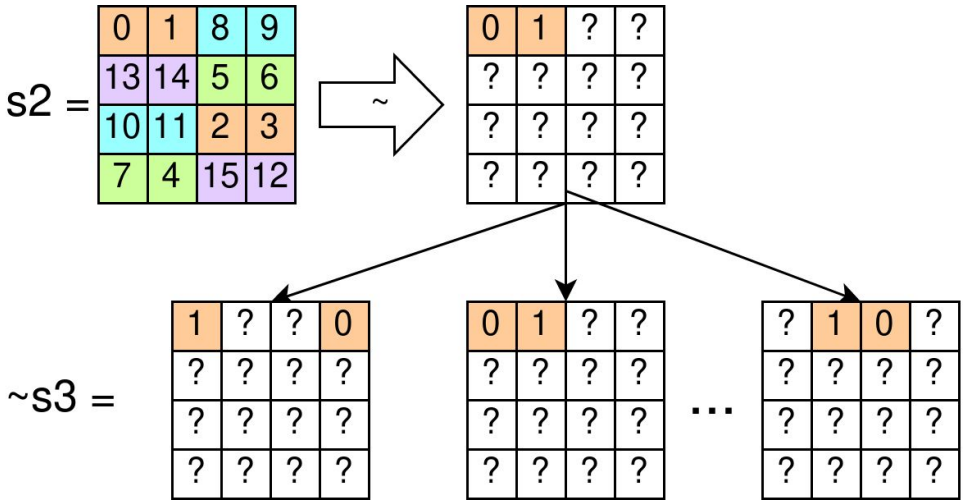
- Exploited graphs for each Reaches_(g^i, g^j) are the same except for their starting node
- Also have nice layers - one for each statement in the program
- So, compute each layer, then compute all-source, all-target reachability with matrix multiply
- Fast and cacheable - key to our efficiency
- **TODO: I could've sworn I had a good picture of this**

Copy count test

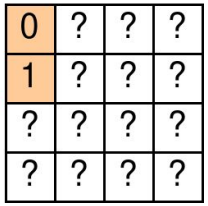
- Problem with pair-reachability: can't catch extra copies of values
  - Ex. conditions on add should've made something 0 but didn't
- Working backwards, compute lower and upper bounds on how many times the value of v^i can be in the output
  - For gathers: bounds on inputs are sums of bounds or outputs that read from there - min/max over set of functions
  - For folds: Each input collected is copied as many times as output
- To test: for each *a* in universe(stmt),, sum up min/max bounds on each v^i equal to *a*, make sure actual number of *a*s needed is in bounds
- **TODO: too late at night to make it a good picture**

Viability tests

Abstract execution?



but not



One idea: only track a subset of locations, see if the terms there can go where they're needed

- Too slow — can't get enough to be useful, even with caching
- Adjusting coarseness of abstraction didn't help
- But …

Choice of abstractions

- With one-location reachability, benchmarks time out
- Three-location data does not fit in RAM for larger benchmarks
- Copy count is necessary
  - Without it, large polynomial multiply benchmark goes from 3 seconds to timeout
- **Note from audience: move this up**

Q2: Abstraction effectiveness

- In many cases, close to optimal pruning
- Deviations
  - Combining multiple inputs (polynomial, weights in convolution)
  - HVX — later instructions could fix mistakes (abstractly)